Support Vector Machine and k-Nearest Neighbors

Gerardo De la O

2022

Real World Example - Classification Algorithms

I currently work at a technology company specializing in conversational analytics. From a a business perspective, one of the most common reasons why other companies engage with us is to improve their customer experience (CX). A very impactful use case of classification models in our industry is predicting customer churn (when a customer stops doing business with a company). If we could tell our client which customers are more likely to churn, why their customer base in general is churning, and which of their front-line employee behaviors affect the outcome, then we will have provided a solution that both improves customer experience and retains revenue.

The process of feature selection will eventually lead to a good, actionable model. In general, predictors that we usually consider are:

```
1.  Reason for call (What the customer said they're calling about)
2.  Indicators of negative experience (repeat interactions, frustration, confusion, etc.)
3.  Positive agent behaviors (advocacy, acknowledgement, issue resolution, etc.)
4.  Negative agent behaviors (powerless to help language, transfers, etc.)
5.  Customer metadata (tenure, products purchased, survey responses, etc.)
```

SVM and KNN Implementation

In this section, we will explore the use of Support Vector Machine and K-Nearest Neighbors classification algorithms to classify credit card application data.

Preparation

```r
library(kernlab)
library(kknn)
library(caret)
library(dplyr)

filename = 'credit_card_data.txt'
data <- read.table(paste0('data 2.2/',filename))
```

SVM with no kernel (linear)

For our linear svm, we iterated through a set of C values and captured the accuracy of each model. Based on the results, it is clear that there is no discernible difference in accuracy between C values of 50-1000, so we choose the value of C=100 given as default.

Parameters for ksvm:

- type = class of prediction. C-svc = classification
- kernel = function used in training and predicting. 'vanilladot' = linear, 'rbfdot' = gaussian kernel
- C = cost of constraints violation. Equivalent to 'lambda' regularization term in the Lagrange formulation

```
x <- as.matrix(data[,-ncol(data)])
y <- as.factor(data[,ncol(data)])

cOptimum = data.frame(c=integer(0), accuracy=numeric(0))

for(i in seq(50,1000,50)){
    cOptimum[i,'c'] = i
    output = data[,ncol(data), drop=FALSE]

    model = ksvm(x, y, type = 'C-svc', kernel='vanilladot', C=i,
                 kpar = list(), scaled=TRUE, class.weights=NULL, prob.model=FALSE)

    predictions = as.factor(fitted(model))

    evaluation <- output %>%
        mutate(model_pred = predictions, accurate= 1*(model_pred == V11))
    accuracy = sum(evaluation$accurate/nrow(output))
    cOptimum[i,'accuracy'] = accuracy


}


cOptimum = cOptimum[complete.cases(cOptimum), ]
rownames(cOptimum) <- NULL

cOptimum
```

```
##       c  accuracy
## 1    50 0.8639144
## 2   100 0.8639144
## 3   150 0.8639144
## 4   200 0.8639144
## 5   250 0.8639144
## 6   300 0.8623853
## 7   350 0.8639144
## 8   400 0.8623853
## 9   450 0.8623853
## 10  500 0.8639144
## 11  550 0.8623853
## 12  600 0.8623853
## 13  650 0.8623853
## 14  700 0.8623853
## 15  750 0.8623853
## 16  800 0.8623853
## 17  850 0.8623853
## 18  900 0.8623853
## 19  950 0.8623853
## 20 1000 0.8623853
```

SVM Results The final model equation and its performance parameters are shown below. Each 'V' coefficient will be multiplied by the corresponding input variable, and a0 is our constant term. A confusion matrix containing more detailed performance parameters is also included to give a more robust picure of model performance.

```r
x <- as.matrix(data[,-ncol(data)])
y <- as.factor(data[,ncol(data)])

model1 <- ksvm(x, y, type = 'C-svc', kernel='vanilladot', C=100,
               kpar = list(), scaled=TRUE, class.weights=NULL, prob.model=FALSE)

#outputs of final model
x = xmatrix(model1)[[1]]
c = coef(model1)[[1]]

#Ax + a0 = 0 (our linear equation)
a <- colSums(x * c)
a0 <- b(model1)

a['a0'] <- a0
a
```

```
##            V1            V2            V3            V4            V5 
## -0.0010065348 -0.0011729048 -0.0016261967  0.0030064203  1.0049405641 
##            V6            V7            V8            V9           V10 
## -0.0028259432  0.0002600295 -0.0005349551 -0.0012283758  0.1063633995 
##            a0 
## -0.0815849217
```

```r
pred = as.factor(fitted(model1))
confusion <- confusionMatrix(data=pred, reference = y)
confusion
```

```
## Confusion Matrix and Statistics
## 
##           Reference
## Prediction   0   1
##          0 286  17
##          1  72 279
##                                           
##                Accuracy : 0.8639          
##                  95% CI : (0.8352, 0.8893)
##     No Information Rate : 0.5474          
##     P-Value [Acc > NIR] : < 2.2e-16       
##                                           
##                   Kappa : 0.7297          
##                                           
##  Mcnemar's Test P-Value : 1.041e-08       
##                                           
##             Sensitivity : 0.7989          
##             Specificity : 0.9426          
##          Pos Pred Value : 0.9439          
##          Neg Pred Value : 0.7949          
##              Prevalence : 0.5474          
##          Detection Rate : 0.4373          
##    Detection Prevalence : 0.4633          
##       Balanced Accuracy : 0.8707          
##                                           
##        'Positive' Class : 0               
## 
```

SVM with kernel (gaussian)

Much like the dot product, we can use kernels as a measure of similarity between 2 vectors. The interesting thing about kernels is that they allow us to work on a higher-dimensional space without explicitly building a higher-d representation of our input (this is called the 'kernel trick'). Conceptually, a kernel svm will generate a linear split in the higher-d space, but when converted back to the original space, the decision boundary will be non-linear!

SVM with Kernel Function - Results The svm with gaussian kernel below performs considerably better than our linear svm. In fact, if we keep increasing the value of C, we can achieve an almost perfect classification of 99.5%! However, given that our decision boundary is no longer linear, we run the risk of over-fitting. Normally, we would use validation and testing to find a good value of C without over-fitting, but since that is not covered in this homework, I chose C = 500 for an accuracy of 97%.

```
x <- as.matrix(data[,-ncol(data)])
y <- as.factor(data[,ncol(data)])

model2 <- ksvm(x, y, type = 'C-svc', kernel='rbfdot', C=500,
               scaled=TRUE, class.weights=NULL, prob.model=FALSE)

pred = as.factor(fitted(model2))

confusion <- confusionMatrix(data=pred, reference = y)
confusion
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction   0   1
##          0 354  11
##          1   4 285
##
##                Accuracy : 0.9771
##                  95% CI : (0.9625, 0.9871)
##     No Information Rate : 0.5474
##     P-Value [Acc > NIR] : <2e-16
##
##                   Kappa : 0.9536
##
##  Mcnemar's Test P-Value : 0.1213
##
##             Sensitivity : 0.9888
##             Specificity : 0.9628
##          Pos Pred Value : 0.9699
##          Neg Pred Value : 0.9862
##              Prevalence : 0.5474
##          Detection Rate : 0.5413
##    Detection Prevalence : 0.5581
##       Balanced Accuracy : 0.9758
##
##        'Positive' Class : 0
##
```

K-Nearest Neighbors

For this exercise, we apply knn algorithm to each row of the data,while using the rest of the data as our training parameters ("anything but" logic). In order to find the best value of k, we tested different k values in increments of 10 across the full span of our data set. While this process is computationally expensive and circumvents easier procedures, it gives us a very clear picture of how different values of k affect the model. Note that a "rectangular" kernel indicates the standard, un-weighted knn referred to in the lecture.

Parameters for kknn:

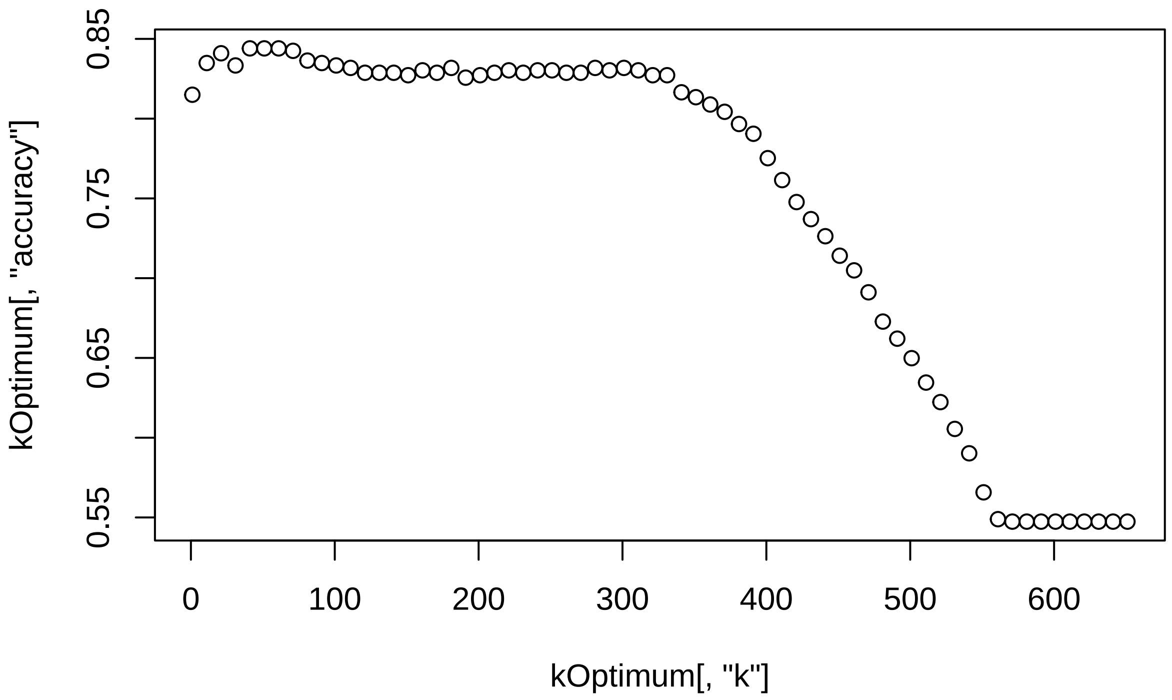
- train = Matrix or data frame of test set cases.
- test = Matrix or data frame of training set cases.
- k = Number of neighbors considered.

```
kOptimum = data.frame(k=integer(0),accuracy=numeric(0))

for(j in seq(1,nrow(data),10)){
    kOptimum[j,'k'] = j
    output = data[,ncol(data), drop=FALSE]

    for(i in 1:nrow(data)){
        train = data[-i,]
        test = data[i,]
        val = kknn(V11~., train=train,test=test,
                     k=j, kernel="rectangular", scale='TRUE')$fitted.values
        output[i,'fitted_val'] = val
    }
    evaluation <- output %>%
        mutate(model_pred = 1*(fitted_val > .50), accurate= 1*(model_pred == V11))
    accuracy = sum(evaluation$accurate/nrow(output))
    kOptimum[j,'accuracy'] = accuracy
}
kOptimum = kOptimum[complete.cases(kOptimum), ]
rownames(kOptimum) <- NULL

head(kOptimum,10)
```

```
##     k  accuracy
## 1   1 0.8149847
## 2  11 0.8348624
## 3  21 0.8409786
## 4  31 0.8333333
## 5  41 0.8440367
## 6  51 0.8440367
## 7  61 0.8440367
## 8  71 0.8425076
## 9  81 0.8363914
## 10 91 0.8348624
```

KNN Results The maximum accuracy occurs relatively early in the iteration process. In fact, by looking at the result series, we see that the maximum occurs at a k between 40-60, with an accuracy of 84.4%. The fit degradates as k becomes too high, indicating bias. Finally, we note note that using other kernels may yield slightly better performance.

```
plot(kOptimum[,'k'],kOptimum[,'accuracy'])
```


0.85
0.75
0.65
0.55
kOptimum[, "accuracy"]
0
100
200
300
400
500
600
kOptimum[, "k"]